APPENDIX H

```
//
//
//  ------------------------------------------------------------------------
//  |----------------------------------------------------------------------|
//  |                                                                      |
//  |   FILE:   hypolist.hpp                                               |
//  |   FUNCTIONALITY: implementation of template functions                |
//  |   PROGRAM:  file to field                                            |
//  |   COMMENTS: retrieval of file from regional server                   |
//  |   AUTHOR:  A. CHRISTIAN TAHAN                                        |
//  |   DATE FIRST VERSION:  02/13/00                                      |
//  |----------------------------------------------------------------------|
//  ------------------------------------------------------------------------


#ifndef _HYPOLIST_HPP_
#define _HYPOLIST_HPP_

template<class T>
Boolean SparseList<T>::Has_No_Kids(t_ptr node) const
        {
        //dummy definition so that at least one instance of
        //ImpObjectList<T> and linker can find the member function of the
class
        ImpObjectList<T> dummy;

        return((Boolean)(List[node].num_kids==0));
        }


template<class T>
SparseList<T>::SparseList(natural chunk)
        {
        chunk_size=chunk;
        free_list=0;
        start_list=0;
        dim_free=0;

        //allocate the nihil=0 element this can't be used
        List.Destroy_And_ReDim(1);

        return;
        }


template<class T>
```

```
void SparseList<T>::Restart()
        {
    free_list=0;
        start_list=0;
        dim_free=0;

        //allocate the nihil=0 element this can't be used
        List.Destroy_And_ReDim(1);
        return;
        }


template<class T>
void SparseList<T>::Reset()
        {
        free_list=0;
        start_list=0;
        dim_free=0;

        return;
        }


template<class T>
void SparseList<T>::Allocate_Mem()
        {
        natural i;
        t_ptr temp=List.Dim();

        List.Save_And_ReDim(chunk_size+temp);

        //link the node of free list
        for (i=temp; i<temp+chunk_size-1; i++)
                List[i].link=i+1;

        List[List.Dim()-1].link=free_list;
        free_list=temp;
        dim_free+=chunk_size;

        return;
        }


template<class T>
t_ptr SparseList<T>::Create(const T & info,t_ptr parent)
        {
```

```
    t_ptr temp;

        if (free_list==0)
                Allocate_Mem();

        //get one node from free list
        temp=free_list;
        free_list=List[temp].link;
        dim_free--;

        //verify that free_node is really free
        Assert(List[temp].num_kids==Node::Kids_Of_Free_Node());
        List[temp].link=parent;
        List[temp].num_kids=0;
        List[temp].info=info;
        List[parent].num_kids++;

        return temp;
        }


template<class T>
t_index SparseList<T>::Num_Node() const
        {
        return (List.Dim()-dim_free-1);
        }


template<class T>
t_ptr SparseList<T>::Next(t_ptr son) const
        {
        Assert (son>0);
        return List[son].link;
        }


template<class T>
void SparseList<T>::Destroy_Node(t_ptr node)
        {
    Assert(node>0);

        if (List[node].num_kids>0)
                merr<<"Attempt to destroy referenced node";

        //decrease parent's num_kids
        List[Next(node)].num_kids--;
```

```
        //add to free list
        List[node].link=free_list;
        free_list=node;
        List[free_list].num_kids=Node::Kids_Of_Free_Node();
        dim_free++;

        return;
        }


template<class T>
void SparseList<T>::Backtrack_From(ImpObjectList<T> & sequence, t_ptr node)
        {
        t_index i=0;

        Assert(node>0);
        sequence.Reset();
        do      {
                sequence.Save_And_ReDim(i+1);
                sequence[i]=(*this)[node];
                i++;
                node=Next(node);
                }
        while ( node!=0);

        // eliminate phantom node
        // is the following instruction necessary in order to eliminate
        // silence and duplicated typo?
        sequence.Save_And_ReDim(i-1);
        T temp;
        for (i=0; i<sequence.Dim()/2; i++)
                {
                temp=sequence[i];
                sequence[i]=sequence[sequence.Dim()-i-1];
                sequence[sequence.Dim()-i-1]=temp;
                }

        return;
        }


template<class T>
void SparseList<T>::Destroy_Branch(t_ptr node)
        {
        t_ptr temp;
        Assert(node>0);
```

```
        if (List[node].num_kids>0)
                merr<<"Attempt to destroy referenced node";

        do      {
                temp=Next(node);
                Destroy_Node(node);
                node=temp;
                }
        while (List[node].num_kids==0 AND node!=0);

        return;
        }


template<class T>
T & SparseList<T>::operator[](const t_ptr son)
        {
        Assert(son>0);
        Assert(List[son].num_kids != Node::Kids_Of_Free_Node());

        return List[son].info;
        }


template<class T>
const T & SparseList<T>::operator[](const t_ptr son)const
        {
        Assert(son>0);
        Assert(List[son].num_kids != Node::Kids_Of_Free_Node());

        return List[son].info;
        }


template<class T>
WellTree<T>::~WellTree()
        {
        l_list.Reset();
        leaves_dir.Reset();
        kid_dir.Reset();
        }

template<class T>
void WellTree<T>::Reset()
        {
```

```
        leaves_dir.Reset();
        kid_dir.Reset();
        l_list.Restart() ;
        }


//needed by Viterbi.num_hypotesis
template<class T>
inline t_index WellTree<T>::Num_Elements() const
        {
    return (l_list.Num_Node());
        }


template<class T>
inline t_index WellTree<T>::Kids_Dim() const
        {
        return kid_dir.Dim();
        }


template<class T>
inline t_index WellTree<T>::Leaves_Dim() const
        {
        return leaves_dir.Dim();
        }


template<class T>
void WellTree<T>::ReDim_Leaves_Dir_To(const t_index ix)
        {
        leaves_dir.Save_And_ReDim(ix);

        return;
        }


template<class T>
inline void WellTree<T>::Exchange_Leaves_Indexes(const t_index i,
                                                                    const
t_index j)
        {
    t_index aux;
        aux=leaves_dir[i];
        leaves_dir[i]=leaves_dir[j];
        leaves_dir[j]=aux;
```

```
        return;
        }


template<class T>
Boolean WellTree<T>::Check_Kid_Presence_And_Get_Num(const T & act_kid,
                                                    p_kid & kid_idx)
        {
        t_index i=0;
        t_index kid_num;

        kid_num = kid_dir.Dim();
        if (kid_num==0)
                return (Boolean) FALSE;
        else{
                while (i<kid_num AND act_kid!=l_list[kid_dir[i]])
                        i++;

                if (i==kid_num)
                        return (Boolean)FALSE;
                else{
                        kid_idx=i;
                        return (Boolean)TRUE;
                        }
                } // end of else

        }


template<class T>
inline const T& WellTree<T>::Get_Leaf_Info(const p_leaf leaf)const
        {
        return (l_list[leaves_dir[leaf]]);
        }

template<class T>
inline T& WellTree<T>::Get_Leaf_Info(const p_leaf leaf)
        {
        return (l_list[leaves_dir[leaf]]);
        }

template<class T>
inline const T& WellTree<T>::Get_Kid_Info(const p_kid kid)const
        {
        return (l_list[kid_dir[kid]]);
```

```
        }

template<class T>
inline T& WellTree<T>::Get_Kid_Info(const p_kid kid)
        {
        return (l_list[kid_dir[kid]]);
        }

template<class T>
inline void WellTree<T>::Create_First_Leaf_Of_Tree(const  T& info)
        {
        //if no elements in tree create leaf
        Assert(l_list.Num_Node()==0);
        leaves_dir.Destroy_And_ReDim(1);
        //0 pointer is NULL
        leaves_dir[0]=l_list.Create(info,0);

        return;
        }


template<class T>
inline void WellTree<T>::Add_Kid_To_Leaf(const  T& info, p_leaf leaf)
        {
        //if no elements in tree create a kid
        if (l_list.Num_Node()==0)
                {
                kid_dir.Destroy_And_ReDim(1);
                //0 pointer is NULL
                kid_dir[0]=l_list.Create(info,0);
                return;
                }

        //abort if tree not empty and no leaves
        Assert(l_list.Num_Node()>0 AND leaves_dir.Dim()!=0);

        //abort if more than one well created
        Assert(l_list.Num_Node()>0 AND leaves_dir[leaf]!=0);

        t_index kid_dim=kid_dir.Dim();
        kid_dir.Save_And_ReDim(kid_dim+1);

        kid_dir[kid_dim]=l_list.Create(info,leaves_dir[leaf]);
        return;
        }
```

```
template<class T>
void WellTree<T>::Prune_All_Dead_Leaf()
        {
    t_index i;
        t_index num_leaves=leaves_dir.Dim();

        // here its not necessary to update leaves_dir
        // since next_gen follows
        for (i=0; i<num_leaves; i++)
                if (l_list.Has_No_Kids(leaves_dir[i]) )
                        Prune_Blind_Branch_From_Leaf(i);

        return;
        }

template<class T>
inline void WellTree<T>::Prune_Blind_Branch_From_Leaf(p_leaf leaf)
        {
        Assert(leaves_dir.Dim()>=1);

        l_list.Destroy_Branch(leaves_dir[leaf]);

        return;
        }


//Backtrack_from(a_node) returns a new list with every element
//containing address of every nodes along path sequence
template<class T>
inline void WellTree<T>::Backtrack_From(ImpObjectList<T> & sequence,p_leaf
leaf)
        {
        l_list.Backtrack_From(sequence,leaves_dir[leaf]);
        }

//start the next generation transform kid_dir leaves_dir;
template<class T>
inline void WellTree<T>::Next_Gen() // leaves=kid
        {
        leaves_dir=kid_dir;
        kid_dir.Reset();
        return;
        }

template<class T>
```

```
inline void WellTree<T>::Subst_Old_Kid_Destroy_Old_Branch_Ins_New(p_kid
old_kid,
                                        const  T & info,p_leaf new_father)
        {
        l_list.Destroy_Node(kid_dir[old_kid]);
        kid_dir[old_kid]=l_list.Create(info,leaves_dir[new_father]);

        return;
        }

#endif
```